MATLAB Tutorial

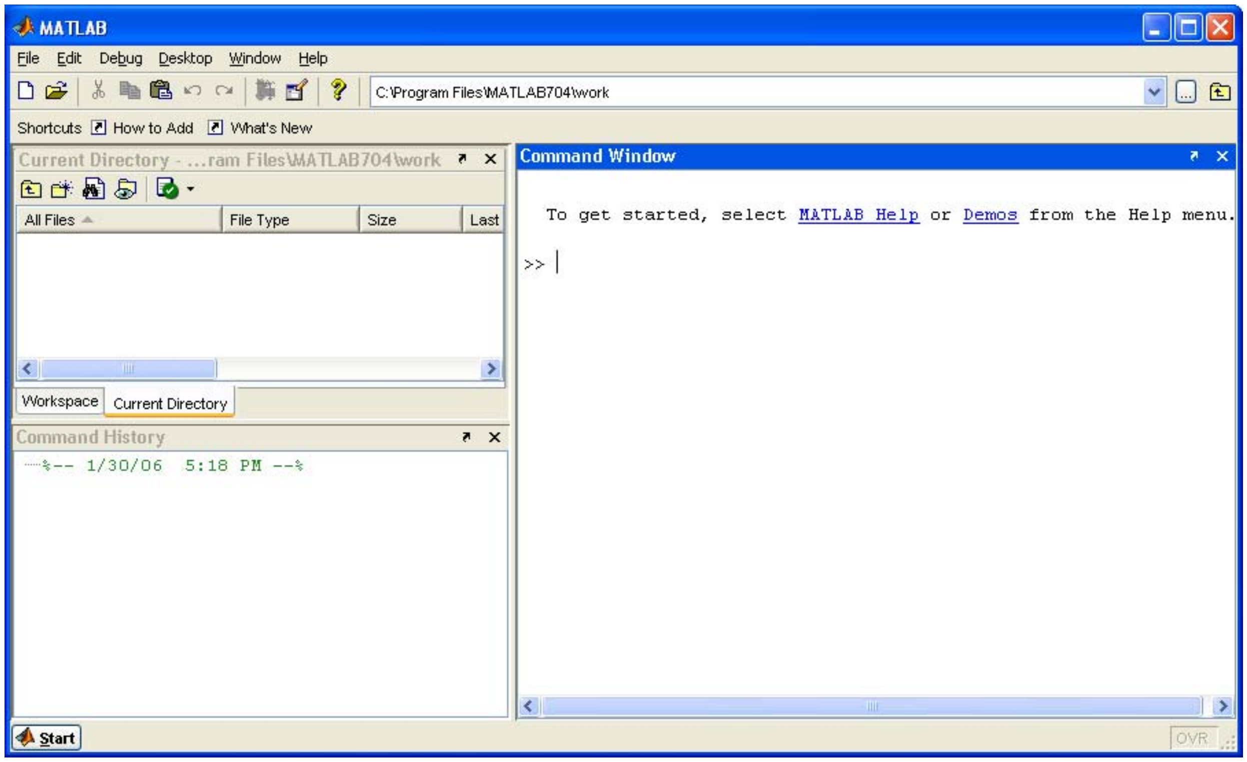
This tutorial is intended to get you acquainted with the MATLAB programming environment. It is not intended as a thorough introduction to the program, but rather as a practical guide to the most useful functions and operations to be used in this course. For additional references and to see a list of some of the many wonderful people who generously contributed to this tutorial, see "Acknowledgments and References" at the end of this document.

I. What is MATLAB?

MATLAB is a programming environment for working with numerical data and, to a lesser extent, symbolic equations. MATLAB uses an interpreted language, which means the code is compiled (translated from what you type into machine-readable code) and run as you type it in. For this reason, it is an easy environment in which to perform a few manipulations on some data and plot the output without having to include a lot of the basic declarations required by more traditional programming languages. In addition, MATLAB contains a vast array of built-in functions for performing manipulations on data.

II. Launching MATLAB

When you open MATLAB, the following screen (or something very similar, different versions looks slightly different but have nearly identical functionality) should appear:


MATLAB
File Edit Debug Desktop Window Help
C:\Program Files\MATLAB704\work
Shortcuts How to Add What's New
Current Directory - ...ram Files\MATLAB704\work
All Files
File Type
Size
Last
Workspace
Current Directory
Command History
Command Window
Start
OVR

```
To get started, select MATLAB Help or Demos from the Help menu.

>> 
```

```
%-- 1/30/06  5:18 PM --%
```

The most important of these is the **Command** window, where you will type all instructions to MATLAB.

The **Workspace** window lists all variables and their sizes and class (array, string, etc.). This window can be useful in debugging (i.e. finding the errors in) your code.

The **Command History** window tells you the most recent commands. You can double-click on a previous command to run it again (to save you re-typing).

You can open or close these windows by choosing the items from the **Desktop** menu.

The **Current Directory** is displayed in the long horizontal box along the top toolbar of the program. This is the default directory that MATLAB uses when you save your work or open files. You should set this to the directory (e.g. the Desktop or a folder) in which you plan to locate your work. You can set this directory by clicking the "**...**" button to the right of the Current Directory box and choosing an appropriate location. This information can alternatively be set using the Current Directory window that appears as a tab in the same space as the Workspace window.

III. Using MATLAB as your calculator

At the most basic level, MATLAB can be used as your calculator with +, –, *, and / representing addition, subtraction, multiplication, and division respectively.

Let's play a bit with this (type along when you see the prompt ">>"):

```
>> 2+2

ans =

     4

>> (2*(1+5))/3

ans =

     4
```

"ans=" is MATLAB shorthand for "the answer is…". If you are not sure about the order of operations, it is always safe to be explicit by using extra parentheses!

There are also very many built-in functions, e.g.

```
>> sin(pi/2)

ans =

     1
```

Note that "pi=3.14159…" is known by MATLAB and that MATLAB is *case-sensitive* (try typing Sin(pi/2) and see what happens). Other useful functions are **exp** [the exponential function], **log** [the natural logarithm], and **log10** [logarithm in base 10].

IV. Assigning variables

A *variable* is a storage space in a computer for numbers or characters. The equal sign "=" is used to assign a numerical value to a variable:

At the command prompt, type:

```
>> a = 5
```

You've just created a variable named `a` and assigned it a value of 5. If you now type

```
>> a*5
```

you get the answer 25.

Now type

```
>> b = a*5;
```

You've just made a new variable `b`, and assigned it a value equal to the product of the value of `a` and 5. By adding a **semicolon**, you've suppressed the output from being printed. Suppressing the output will be important later. To retrieve the value of `b`, just type

```
>> b
```

without a semicolon.

Now try typing

```
>> a = a + 10
```

This re-assigns to the variable `a` the sum of the previous value of `a` (which was 5), plus 10, resulting in the new value of the variable `a` being 15. *Note: this illustrates that '=' means 'assign' in MATLAB. It is not a declaration that the expressions on the two sides of the '=' sign are 'equal' (because certainly* `a` *cannot equal itself plus 10!). We will see later that a double equal sign "==" is used when we want to test if two quantities are equal to each other.*

V. Vectors and Matrices

In MATLAB, all numerical variables are real-valued matrices, or *arrays* (of type "double" for the programmers out there). This makes it easy to perform manipulations on groups of related numbers at the same time. You did not realize it, but the variables you created above are 1 row by 1 column (or "1x1" for short) matrices. To confirm this, click in the Workspace panel to make it active, then in the main MATLAB menus choose **View**>>**Choose Columns**>>**Size** to see that the variables you created above such as `a` and `b` are actually 1 x 1 arrays.

Let's see how more about how arrays work. Type

```
>> a = [1 2 3 4 5]

a =

     1     2     3     4     5
```

You have just created an array containing the numbers 1 through 5. This array can be thought of as a matrix that is 1 row deep and 5 columns wide (often called a *row vector*). To confirm this, use the **`size`** command which tells you the number of rows (first element returned) and number of columns (second element returned) in an array:

```
>> size(a)

ans =

     1     5
```

To obtain the number of elements in a vector, use the **length** command:

```
>> length(a)

ans =

     5
```

You can also add or multiply arrays by a constant, or add arrays. Try:

```
>> b = 2*a

b =

     2     4     6     8    10

>> b+a

ans =

     3     6     9    12    15
```

You can access the value of a given element of `a` or `b` by using parentheses. Type

```
>> a(3)
>> b(4)
```

and you see the values of the 3rd element of the `a` array and the 4th element of the `b` array, respectively. You can also use `b(end)` or `a(end)` to see the last element.

Finally, you can *append* one vector onto another to make a longer vector. For example, try typing:

```
>> a = [a 8 9 10]

a =
     1     2     3     4     5     8     9    10
```

This re-assigns the variable a to equal a new row vector containing the elements of the previous vector `a`, followed by the new elements [8 9 10].

If you prefer to arrange your array into a column (i.e. a 5 row x 1 column matrix, or *column vector*), you can do this by separating the elements by semicolons:

```
>> a = [1; 2; 3; 4; 5]

a =

     1
     2
     3
     4
     5

>> size(a)

ans =

     5     1
```

Now suppose we decide that we really would have liked to organize the elements of a into a row vector. There are two ways to get back to our original assignment (well… 3 ways if you include just re-typing, but we're far too smart to do that!):

Method 1: use the **uparrow key** (↑) to scroll back through your previous commands until you reach the a = [1 2 3 4 5] command and then press return (equivalently, you also could have used the Command History window). The **downarrow key** (↓) scrolls forward through previous commands (in case you scroll back too far).

For practice, undo this by double-clicking on a = [1; 2; 3; 4; 5] in the Command History window so that a is again a column vector.

Method 2: Another way to change from a row to a column vector is to do the *transpose operation*, denoted by a single quotation mark **'**. The transpose of a matrix exchanges the rows and columns of a matrix.

Let's try it:

```
>> a = a'

a =

     1     2     3     4     5
```

The above line re-assigns to the variable a the values of the row vector a'.

Now let's construct (i.e. create and assign values to) a matrix that we will name "m":

```
>> m = [1 2; 3 4]

m =

   1   2
   3   4
```

Note the syntax: elements in the same row are separated by spaces. A new row is designated by the semicolon.

To find out the value of the element in the 2nd row, 1st column of m, type:

```
>> m(2,1)

ans =

     3
```

Similarly, if we would like to re-assign this element to equal 7 instead of 3 we would type:

```
>> m(2,1) = 7

m =

     1     2
     7     4
```

The transpose of matrix m is found by typing:

```
>> m'

ans =

     1     7
     2     4
```

Notice that the previous 1st row is now the first column and the previous 2nd row is now the second column.

VI. More on Vectors, plus Plotting a Graph

Suppose we want to plot a graph of the function y = x^2 from x = 0 to x=5 with points drawn every 0.5 units along the x-axis (note: ^ is MATLAB's symbol for raising a number to a power).

We could specify the x vector explicitly, writing x = [0 0.5 1.0 1.5 etc.] but this would be quite painful, especially if x were a larger vector.

Instead MATLAB uses the **colon operator** (:) to specify regular sequences of elements. Try typing

```
>> 1:5
```

```
ans =

     1     2     3     4     5
```

We can also increment (or decrement) by an amount other than one by typing, for example:
```
>> 10:-2:4

ans =

    10     8     6     4
```

In general, the syntax for creating arrays in this manner is *start number:size of step:end number*.

Now let's go on to our plotting y=x^2 example. We assign x as:

```
>> x = 0:0.5:5
```

To obtain the squares of the elements of x, one might think we should type x^2 or x*x but either of these (which are equivalent) attempts to multiply two row vectors by each other using the rules of matrix multiplication and this is not a legal mathematical operation:

```
>> x*x
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

In matrix multiplication, the number of columns of the first matrix must equal the number of rows of the second matrix, e.g. (1:3)*(1:3) gives a syntax error but (1:3)*(1:3)' would be ok and gives 14 (try it! Remember that you can use the up arrow to go back to a previous line and revise it). The reason MATLAB says "`Inner matrix dimensions must agree`" is because we can only multiply a M_1 x N_1 element matrix by a M_2 x N_2 if $N_1 = M_2$, i.e. if the inner two of the 4 listed matrix dimensions agree (i.e. are equal). Note that the resulting matrix has size M_1 x N_2 by the rules of matrix multiplication. Thus, a 1 x 3 matrix can't be multiplied by another 1 x 3 matrix but it can be multiplied by the transpose of a 1 x 3 matrix (which has dimensions 3 x 1). If matrix multiplication is unfamiliar to you or if you are just rusty, I recommend brushing up on it.

So how do we take the square of *each element* of a matrix? *To multiply two arrays by each other element by element, you have to add a period before the operator*. For example, the following two statements both make a vector y whose elements contain the squares of the elements of x:

```
>> y = x.*x
>> y = x.^2
```

This is a general MATLAB syntax: always add the **.** before any operation that you would like to conduct element by element (e.g. addition, subtraction, division, etc.).

Great! Now we're ready to plot y vs. x using the **`plot`** command. To do this, type:

```
>> figure(1)
>> plot(x,y)
```

This brings up a window entitled "Figure No. 1" and plots `x` on the horizontal axis and `y` on the vertical axis. All figures are numbered in MATLAB--you can use any integer you want, and if you just type '**`figure`**' it will make up a number for you (specifically, the smallest unused positive integer). If you are only plotting a single figure, you can omit the figure command and MATLAB will automatically draw your plot in Figure No. 1.

If you want to print the figure, you can do that by choosing **File**>>**Print** from within the Figure window. I recommend using **File>>Page Setup** to make sure the window maps onto the paper you're using in a pretty way.

Now let's tackle the only slightly more challenging problem of plotting more than one function at a time. Type:

```
>> hold on
>> plot(x,x,'r')
```

The **`hold on`** command tells MATLAB to hold onto what is currently on the plot if new data is plotted; otherwise, the default behavior is to replace what is there (you can go back to this by typing '`hold off`'). The second function simply plots `x` versus itself in red. The last argument (i.e. item you specify, in this case `'r'`) to the plot function is a *string* (i.e. array of characters) where you can pass some parameters to the plotting function. The **single quotes** is the syntax for telling MATLAB that the r is the character r and not a variable you may have created and named r. In this case, the letter r is an optional argument to the plot function that tells MATLAB to draw the plot in the color red (the default color is blue). To really get into all of the various options is complicated (type '`help plot`' if you want to see), but if you follow the pattern of the examples here you might not need to get into all of the details.

Now we have two plots in the same window. These plots are lines through the points we have given MATLAB. Suppose we want to show the data points. We can do this by adding two new plots. First, close the previous figure window by using **`close(#)`**, where # is the figure number, or by clicking the close box on the window itself. Then type:

```
>> plot(x,y,'o')
>> plot(x,x,'ro')
```

This adds two new plots. The first plots `y` vs. `x` using o's at each point. The second plots `x` vs. `x` using o's at each point and the color red.

Now let's add some labels to the graph and practice changing the axis limits using the **`title`**, **`xlabel`**, **`ylabel`**, **`legend`**, and **`axis`** commands. Type:

```
>> title('First two powers of the integers and half integers')
>> xlabel('Integers and half integers')
>> ylabel('First two powers')
>> legend('second power','first power')
>> axis([0 6 0 50])
```

If you would like more help on the syntax of any of these commands, type 'help *commandname*'.

In this course, you will often be interested in putting more than one plot in a window for clarity and to save paper. This is where the **subplot** command comes in. Close the window you have been working in and now type:

```
>> figure(1)
>> subplot(2,1,1)
>> plot(x,y,'o')
>> title('Integers and half integers squared')
>> subplot(2,1,2)
>> plot(x,x,'o')
>> title('Integers and half integers')
```

`subplot` allows you to define an MxN matrix of plotting panels within a figure, where M=the number of rows of plotting panels within the figure is the first argument of the function, N=the number of columns of plotting panels is the second argument, and the third argument describes which plotting panel to draw in presently. Above, we have defined a 2x1 matrix of plotting panels, and plotted x vs. y in the 1st one and x vs. x in the 2nd one. Note that we specify the title *after* calling the `subplot` and `plot` commands.

VII. Functions

MATLAB has many built-in functions that allow many operations to be carried out in a single line. You just learned about a few (e.g. `size`(*matrix*) and `plot`(*vector1,vector2,formatting*)). The general format is *functionname*(*arguments*) with the appropriate function name and arguments. For example, in the course we will often create an array t of time points:

```
>> t=1:0.25:6;
```

Then,

```
>> x=sin(t)
```

creates a vector of the same size as t, with values of the sine of t at each point. (try plotting x=sin(t) vs. t to check this out).

Here's another useful one:

```
>> y=zeros(1,5)
```

The **`zeros`** function assigns to the variable y a 1 x 5 array of zeros.

VIII. Logical structures: loops, the if statement, and logical operators

A) For loops

Now that you can perform simple operations, let's learn how to group operations together using computer logic. The first structure that will be useful is known as the "*for loop*". The **for** command is useful when we would like to perform the same operation or set of commands many times. For example, suppose we wanted to know the square of the numbers 1, 4, and 5. We could type 1^2, followed by 4^2, followed by 5^2, but this would clearly be quite tedious (especially if we wanted to know the square of far more numbers). A `for` loop is perfect for this type of situation because we are performing the same operation (squaring a number) over and over again. The syntax for accomplishing this is the following. Type:

```
>> for A=[1 4 5]
      disp('You will see the current value of A^2 below:')
      A^2
   end
```

The output you should see in response to this is:

```
You will see the current value of A^2 below:

ans =

     1

You will see the current value of A^2 below:

ans =

    16

You will see the current value of A^2 below:

ans =

    25
```

Notice what happened: MATLAB ran the two lines "disp('You will…')" and "A^2" three separate times, once for each of the elements in the vector [1 4 5]. Note that the **`disp`** command prints to the screen whatever its argument is (in this case, a string of text). The above example also illustrates the syntax of a `for` loop, namely the first line of a `for` loop is "`for` *variable* = *vector*", the last line is "end", and the in-between lines are the *commands* to be executed over and over again, once for each element of the vector.

Specifically, what the `for` loop does is:

1. Assign the *variable* (e.g. A) to equal the first element of *vector* (e.g. 1 if the vector is [1 4 5]).

2. Execute all lines following the "for *variable* = *vector*" line and before the "end" line.

3. "Loop" back to the beginning, now assigning *variable* (e.g. A) to equal the 2nd element of *vector* (e.g. 4 in the example above), and again execute all lines before the "end" line.

4. Keep looping through the elements of *vector* until all values have been used. When the last element of *vector* (e.g. 5 in the example above) has been assigned to *variable* and the lines before the "end" line have been executed, the loop is finished executing.

We can also use `for` loops to assign values to the elements of a matrix. If we want to make a vector `y` containing the values of sin(x) from x=0 to x=pi in steps of pi/10, we could type:

```
>> clear y
>> x=0:pi/10:pi
>> for i=1:length(x)
      y(i) = sin(x(i))
   end
```

This program has the same effect as typing y = sin(x), but illustrates how for loops work. It first clears the previous value of `y` from memory with the command **`clear`** `y`. It then creates the vector `x`. Then, it cycles through the loop once for each value of the array `1:length(x)` (which has values 1 through 11 since the length of the vector x is 11) and, for each value of `i`, it assigns to the ith element of array `y`, the sine of the ith element of array `x` (e.g. y(1) = sin(x(1)) = sin(0) = 0 because the first element of array `x` is x(1)=0.) Check this by typing:

>> y(1)

Notes:
1) The above code typically runs much slower than if we were to use the vector assignment operations we've learned so far (like y=sin(x)). For this reason, using vector operations is preferable to using for loops when possible. However, the above is a nice example of the use of the for loop and is similar to the types of for loops we will use next week so make sure you understand it.

2) *Sometimes you may find that you have made a mistake where the computer appears to freeze because it is looping through a near-infinite number of iterations of a `for` loop (or appears to freeze for some other reason you can't figure out!). If this happens, you can interrupt your simulation and return to the prompt in the Command window by typing **CTRL-C**.*

B) If statement

The other most useful logical operation is the "**if**" statement. Try:

```
>> for A=1:5
         if (A > 3)
             A
        end  %end of if statement
end %end of for loop
```

This program steps through the values in the array 1:5 and, if the value of A during the current step is greater than 3, prints this value of A. Note that for loops and if statements always need to be concluded with an end statement. Also note the **%**: anything following the % sign is a *comment* (i.e. note to yourself) and is ignored by MATLAB. ***Commenting your code is tremendously useful, as it allows you to remember the logic of your code when you look back at it. It also is easier for others who may read your code to follow your logic. You should use comments liberally.***

C) Boolean (true or false) logic

Often we simply want to know if something is true or false. MATLAB denotes true by the integer 1 and false by the integer 0. For example, try:

```
>> a=1:6;
>> b = (a>2)
```

The above line assigns to b a vector of the same size as a with 0's (false's) and 1's (true's) designating whether that element of a was greater than 2. You can also do < (less than), <= (less than or equal to), >= (greater than or equal to), == (equal to; *note that we use the double equal sign for the logical operation of comparing two values. The single equal sign is only used for assigning values to variables. See the example in the next section*).

D) While loop

Another type of loop, similar to the for loop, is called the **while** loop. If you are interested, you can get more information about this by typing 'help while' at the command prompt or by consulting the Pratap book.

IX. More logic, and M-files

Up until now, you've typed all of your commands directly into the command line. Of course, for doing your homework, you'll probably want to put your code into a file so you can print it out easily and remember what you have done. Let's make a MATLAB code file, or "*m-file*", for the last example.

In the MATLAB command window, choose **File>>New...M-file** from the File menu. Now you should have an open text window. Re-type the following lines into the file:

```
for A=1:5
  if (A > 3)
    A
  end  %end of if statement
end %end of for loop
```

Now choose **File>>Save**. Save it to the Desktop (or make a folder for yourself on the desktop and save it there) as 'Lab1a.m' or something like that, where the ".m" at the end of the filename tells MATLAB that this is a MATLAB program. *Note: do not use spaces or special characters in file names (the underscore character '_' is sometimes helpful in place of a space).* Now, before running your m-file, make sure you are working in the same directory as your file by checking the Current Directory in the toolbar. If the Current Directory is not set to the directory where your file is, change it now.

Now you are ready to run your m-file. Type 'Lab1a' and the commands should run. If you ever want to run an m-file without being in that directory, see 'help **addpath**'. Alternatively, you can both save what you have typed and run your code simultaneously by clicking the button in the toolbar which is equivalent to **Debug**>>**Run** (or **Save and Run**, or the **F5** key).

Let's add a bit more logic to our loop. Revise your m-file to read:

```
clear s;  %clears the variable s out of memory
for A=1:5
  if (A > 3)
    A             %print out the value of A
  elseif (A==3)  %note syntax: "A equals 3" is denoted by a double equal sign!
                 %the single "=" is used only for assigning values to variables.
                 %Confusing these is one of the easiest ways to have an error in your code!
    s = 'hello'   %to print and/or assign a text string, put it inside single quotation marks
  else           %A equals 1 or 2
     -A          %print out negative of the value of A
  end            %end of if statement
end              %end of for loop
```

Can you predict what this will output? Try it out by saving this file (always make sure to remember to save before running your code—this is a common error!) and running it. The **if…elseif…else** structure is great for going through logic. The elseif or else can be omitted if not needed. Type 'help if' for more information.

The 'clear s' statement at the beginning of this script removes any information that might have been in the variable s from the computer's memory (e.g. if you had previously assigned a variable s earlier in your MATLAB session). **You should always clear all variables from the computer's memory before starting a new program.** Usually, it is simplest to just type **'clear all'**, which removes all variables from memory, at the top of your m-file. To close all open windows (so that you don't end up with plots from old runs appearing in your new figures), type **'close all'** – to be safe, I recommend putting 'clear all' and 'close all' at the top of all m-files you write.

X. Getting help

Learning to use MATLAB help is an essential part of programming in MATLAB, as MATLAB has far more commands than can be contained in any single book. There are many places to get help for MATLAB. Some of them are:

1. Typing '**help** *commandname'* where *commandname* designates the command you want to know the syntax of.

Alternatively (I prefer this one), type '**doc** *commandname*' which brings up the much prettier MATLAB help window.

2. If you do not know the command name, you can type '**lookfor** *ARelatedWordYouCanThinkOf'*. For example, if you wanted to know about sinusoidal functions, you could type 'lookfor sine'. If you get sick of its scrolling through zillions of functions that use sinusoidal functions, you can type **CTRL-C** to interrupt a command and get back to the command prompt.

3. You can also bring up MATLAB's help/search engine by typing **doc** at the command line or by clicking on "**Help**>>**MATLAB (or Product) Help**" in the main MATLAB window. Type words you are looking for into the "**Search**" box (in the left pane of this window) or you can try to look through "**Contents**". In my experience, "Search" is much better than the **lookfor** command.

4. Websites:

MATLAB's webpage:
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml
Quick reference for many MATLAB commands
http://www-psych.nmsu.edu/~jkroger/lab/Manuals/MATLAB_commands.htm
Zillions of other MATLAB tutorials:
http://www.google.com/search?q=matlab+tutorials&ie=UTF-8&oe=UTF-8

Books:
▪ Simple accessible intro: Pratap, R. *Getting Started with MATLAB 7: A Quick Introduction for Scientists and Engineers* (Oxford University Press, 2006).
▪ More advanced and thorough: Hanselman, D. and Littlefield, B. *Mastering MATLAB 6* (Prentice Hall, 2001).

5. When all else fails, type '`why`' at the command prompt as many times as is necessary (type 'help why' for an explanation).

Acknowledgments

Thanks to the following people who generously contributed material to this tutorial:
Yue Hu (Wellesley Physics)
Stephen van Hooser & Larry Abbott (Brandeis University)
Justin Werfel (M.I.T.)
Mark Nelson (Univ. of Illinois)

Solved Exercises **(do these all in separate .m files)**

These exercises should both serve as a math review and get you some more practice using MATLAB. Feel free to consult the solutions (next page) as needed, especially if you are new to programming--it is most important for you to be able to understand how the code in the solutions works.

1. *Assigning functions*
(a) Assign values to, and then plot, the function

$$f(x) = 1 + \sin(x)e^{-x}$$

for values of x ranging from x = 0 to x = 10 in steps of size dx = 0.1. (*Be careful to use the correct type of multiplication.*)

(b) What is the value of *f* at x = 2? What is the second element of the array f in MATLAB. *Be careful here* – this problem illustrates a potential point of confusion:
In math class, f(2) is shorthand for f(x=2), i.e. the value of f when x=2.
In MATLAB, f(2) means "the 2nd element of the array f".

If you want to know what the value of the function f is at x=2, you can do this in two ways (you should try both):

(i) In the Figure window, click on the data cursor icon from the toolbar in the window. Then click on your graph to find the coordinate values of various points (once you've found 1 point, you can move to other points by using the left and right arrow keys on the keyboard).

(ii) Find out which element corresponds to x = 2. You could either think carefully about the size of dx and the fact that you started your array at x = 0 to figure this out manually [and then check that you figured it out correctly by typing "x(*your guess for the correct index*)"].
Alternatively (try this one!), you can use the **find** command by typing the line:

```
>> find(x == 2)
```

This command will return the element (or "index") of the x array corresponding to the value x=2. If you assign this to a variable (e.g. to a variable that you named "Index") as follows:

```
>>Index = find(x == 2)
```

then you can use Index to find the value of f at x=2 by typing `f(Index)`.

2. *Nested for loops*
It is common in programming that we want to repeat over and over again a set of code that has a `for` loop in it. To do this we make a second `for` loop inside of which the first `for` loop sits. We call the first `for` loop the "outer loop" because its `for` and `end` statements contain the second, "inner" `for` loop.

Predict what the output of the code below will produce by working through it with pencil and paper on a clean piece of paper (i.e. try to predict what the computer will output to the screen; in doing this, you may also find it useful to calculate the values of `NumTimesThroughOuterLoop` and `NumTimesThroughInnerLoop` that MATLAB calculates as the program runs but that aren't printed to the screen because of the semicolon). *After doing this*, create an m-file and type in the code below to check your answers.

```
NumTimesThroughOuterLoop = 0;  %this variable will keep track of the number
                               %of times the code has run through the outer loop
NumTimesThroughInnerLoop = 0;   %this variable will keep track of the number of times
                               %the code has run through the inner loop
for i=10:10:30
  NumTimesThroughOuterLoop = NumTimesThroughOuterLoop + 1;
  for j=1:2
     NumTimesThroughInnerLoop = NumTimesThroughInnerLoop + 1;
     i_plus_j = i + j
  end
end
NumTimesThroughOuterLoop
NumTimesThroughInnerLoop
```

3. *Multiplying a vector by a matrix*

Multiply the column vector v=[1; 2] by the matrix A=[2 3; 3 4; 4 5] . We'll call the resulting vector **w**, i.e. **w = A·v**.

i) First, do this matrix multiplication out 'by hand' as a review of matrix math (if you can't remember the formula, it's listed in part (iii), or see Dayan & Abbott Appendix A.1).

ii) Next, do it the easy way by assigning the matrices in MATLAB and taking their product using MATLAB's built-in commands.

iii) Finally, pretend MATLAB didn't have built-in commands for doing matrix multiplication. Instead, write an m-file that implements the matrix multiplication rule:

$$w(i) = \sum_{j=1}^{N} A(i, j) * v(j)$$

where the vector and matrix elements above are written in MATLAB notation, so that w(i) is the i^{th} element of w and A(i,j) is the element located in the i^{th} row and j^{th} column of the matrix A. Also by the rules of matrix multiplication, note that N = (the number of rows of v) = (the number of columns of A). In our case, N=2. Put your final answer in the vector w.

Before programming this, first confirm that the formula above corresponds to the procedure you did when multiplying the matrices 'by hand'. Then, to program this, you will first want to type the command "`clear all`" which will make sure all variables have been cleared. You will then want to assign v and A to the given values above and *initialize* w to be an appropriately dimensioned matrix of zeros before doing the sum (i.e. assign w an initial set of values at the start of your code. This tells MATLAB to set up a placeholder for the w vector so that it's ready to assign values to w when you do your future calculations. Although initialization is not required, doing this makes MATLAB code run much faster so it's a good habit to initialize all unknown matrices to some value such as zero at the beginning of your code).

Finally, use 2 nested for loops. Note that at the conclusion of this program, we would like to have assigned values to w(i) **for** each of its 3 elements, i.e. for i=1, i=2, and i=3. Thus, the outer loop will be over the values of the index i.

To determine any individual value w(i), we can build up its final value one step at a time. First, we initialize w(i) to equal zero (which you did above). Then, we add successive terms in the sum above to it (just as you did when you did the matrix multiplication by hand). That is, we need to keep repeating the operation "add to the current value of w(i) an additional amount A(i,j)*v(j)" **for** each term in the sum above (i.e. for j=1 and j=2 in our case). More generally, if we were multiplying bigger matrices and vectors, we would need to add terms A(i,j)*v(j) for each of the N=length(v) elements of v).

Hint: What line of code in MATLAB increases w(i) by an amount A(i,j)*v(j), i.e. assigns to w(i) its previous value plus an amount A(i,j)*v(j)?

4. *Finding prime numbers*

(a) Determine if the number N = 11 is prime by checking whether it is divisible by 2 or 3 or 4 or ..., up to round(N/2). If it is not prime, display the text 'Not prime'. To check whether N is divisible by the tested divisor, use the mod command (see MATLAB help or doc on this command). Next try this out for the number N = 22 to make sure your code is working.

(b) Modify your code to print 'Is prime' if the number is prime, i.e. if the code makes it through the for loop without triggering 'Not prime' to print. To do this, add a variable called "PrimeFlag" to your code and assign it to equal 1 (denoting "true") at the beginning of your code. *If* N is actually not prime, then this will be discovered by the for loop from part (a) and, when your code first identifies this fact, it should re-assign PrimeFlag to equal 0 (denoting "false"). Also, rather than having any display commands within your for loop, use an if...else type of statement after the for loop to determine whether to display that N is prime or N is not prime.

[*Aside:* If you wanted to be efficient, you might terminate the loop as soon as you find the first divisor that goes into N with no remainder. For example, as soon as you discover that 2 goes into 22 (or any even number), it would be nice to terminate the for loop. This can be done by putting the command "**break**" inside the if statement. When this command is encountered, it immediately terminates the for loop and goes to the first line following the for loop – see the help or doc manuals for more details.]

(c) Modify your code to now loop over N from 2 to 30 and, for each value of N, print out either "[*value of N*] is not prime" or "[*value of N*] is prime". This will require nesting your previous code within a for loop over N (because you want to repeat your previous code over and over again for different values of N).

Clever trick for final output: to display the phrase "[*value of N*] is prime", where [*value of N*] here denotes whatever value of N your loop is currently set at, you unfortunately *cannot* simply type disp('N is prime') because that would tell MATLAB to print the character N (rather than the value stored in the variable N). Instead you can use the command int2str which converts the integer N to the string of text characters representing its value, and then combine this with "is prime" by typing disp([int2str(N) ' is prime']). Here, the term in brackets is interpreted as an array of characters, the first ones of which are the numbers representing the value of N and the latter of which are the text characters ' is prime'.

5. *Numerical calculation of derivatives*

Taking derivatives is easy with MATLAB if we use some clever functions. Recall that the derivative of the function x(t) at time t is defined as:

$$\frac{dx(t)}{dt} = \lim_{\Delta t \to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t}$$

Numerically, we can approximate the derivative by using a very small (although not strictly zero because the computer can't divide by zero) value of Δt this operation is very easy to do:

i) We define a vector t=*t_start:dt:t_end* that is discretized into steps of size dt (dt is really supposed to be the limit as Δt approaches zero, so we really should say "Δt" for the finite step sizes we'll use. However, typing the Δ is difficult so we'll just write 'dt'. This is the convention in numerical simulation programs). **Assign a variable dt=0.1 and then make a vector t that goes from t = –1 to t = 1 in steps of size dt.** You can do this in the command line.

ii) Then we define the vector x(t). For our example, let's consider the function x(t) = sin(2*pi*t). **Create this vector in the command line.**
Potential point of confusion: Be careful not to confuse the parentheses printed in mathematical functions (e.g. x(t) meaning "x as a function of t") and the parentheses used by MATLAB when accessing elements of arrays (e.g. `x(1)` meaning "the first element of the vector `x`"). For example, the first element of the `t` vector, t(1), has value `t(1) = –1`. Similarly, the first element of the x vector we just defined is x(1) = sin(2*pi*t(1)) = sin(-2*pi) = 0.

For concreteness, check this out by typing `x(1)` to see what the value of x is at the first time point and `x(2)` to see what the value of x is at the second time point. **What is the value of x at time t = -0.4 and which element of the array x does this correspond to?**

iii) The derivative rule says that to get dx/dt at any time t, we need to take the difference between x(t+dt) and x(t), and then divide this difference by dt,

e.g. to get the derivative of x at the first time point (t = -1), we would take the difference between the value of x at the 2nd time point, x(2) and the value of x at the 1st time point, x(1), and then divide this difference by dt:
>> **xderiv(1) = (x(2) – x(1))/dt**

We would like to know the derivative not just at the first time point but at all time points, e.g. the derivative of x at the ith time point would be given by:

xderiv(i) = (x(i+1) – x(i))/dt

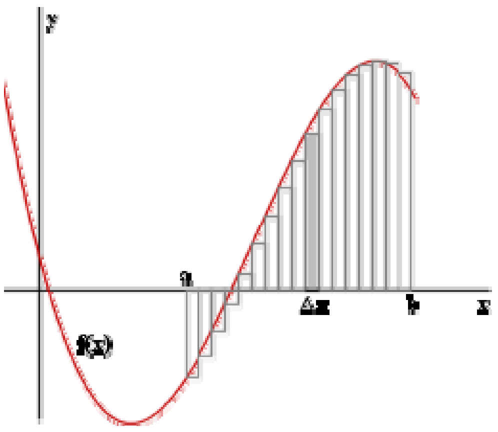
One can calculate differences between elements of a vector for an entire vector all at once using the **diff** command. Type 'help diff' and then try entering diff(x) at the command line.

In a .m file, use the diff command to evaluate the derivative of x = sin(2*pi*t) using the t vector from parts (i) and (ii). Note that diff(x) is 1 element shorter than x because there is no final element (we can't take the derivative of the final point because we don't know the value of x(t+dt) there). When plotting dx/dt vs. t, therefore plot versus a shorter time vector that does not include the last element (*Hint: to access only the first 3 elements of a vector b, for example, you would type* `b(1:3)` *and to access the 2nd-to-last element of b you*

***could type* `b(end-1)` *since* `end` *corresponds to the index of the last element)*. Make 2 figures within the same window, one showing x(t) vs. t and the other dx/dt vs. t.**

Compare to the analytic expression for the derivative (i.e. to the exact formula for the derivative of sin(2πt), which you should calculate). Re-run with dt = 0.01. Can you see why it helps to have dt very small to calculate accurate derivatives?

6. *Computing the value of an integral*
Recall the definition of an integral as the area under a curve, which can be broken into small rectangles of size f(x)*Δx around each point x:



$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sum_{x=a}^{b} f(x)\Delta x$$

That is, to integrate a function we want to step through the values of the function in steps of size Δx and sum the areas of rectangles of height f(x) and width Δx. In MATLAB notation, we wish to put x into a vector starting at x=a, ending at x=b, and discretized into steps of size dx (again, really Δx, but we'll write 'dx' for ease of typing). Thus, in more MATLAB-y notation, we write:

$$\int_a^b f(x)dx = \sum_{i=1}^{N} f(x(i)) * \Delta x = \Delta x * \sum_{i=1}^{N} f(x(i))$$

where N=length(x). Based on the above formula, write a simulation which computes $\int_1^3 x^2 dx$ (and check your answer by doing this numerically). Try dx=0.1, dx=0.01, and dx=0.001 to test the accuracy of your result.

Hint: Although you could program this using a for loop, instead use the command sum to find the sum of the elements of the array f(x(i)), i=1 to N [or try programming it both ways].

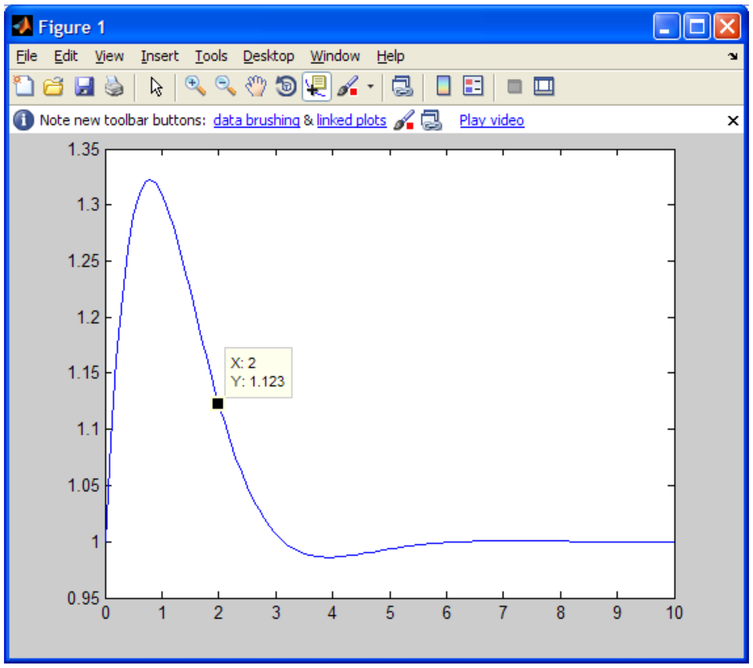
Solutions to Exercises

1. *Assigning functions*

```
clear all;  %clear all variables
close all;  %close all figures

%part (a)
x = 0:0.1:10;
f = 1 + sin(x).*exp(-x);
plot(x,f)

%part (b)
Index = find(x==2)
f(Index)  %value of f when x=2
```



2. *Nested for loops*

```
i_plus_j = 11

i_plus_j = 12

i_plus_j = 21

i_plus_j = 22

i_plus_j = 31

i_plus_j = 32

NumTimesThroughOuterLoop = 3

NumTimesThroughInnerLoop = 6
```

3. *Multiplying a vector by a matrix*

(i)

$$\begin{pmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2*1+3*2 \\ 3*1+4*2 \\ 4*1+5*2 \end{pmatrix} = \begin{pmatrix} 8 \\ 11 \\ 14 \end{pmatrix}$$

(ii)

```
>> v = [1; 2]

v =
     1
     2

>> A = [2 3; 3 4; 4 5]

A =
     2     3
     3     4
     4     5

>> w = A*v

w =
     8
    11
    14
```

(iii) **Notice the commenting and how the for loops are formatted for easy readability:**

```
% Exercise 1, part iii: multiply a vector by a matrix
clear all; %clear all variables from memory
v = [1; 2];
A = [2 3; 3 4; 4 5];

% note that because A is a 3x2 matrix and v a 2x1 matrix,
% j in the sum should go from 1:2 (2 = length(v))
% and w will be a 3x1 matrix (3=# of rows of A, 1=# of columns of v)
% From the help menu, size(A,1) gives the number of rows of A
w = zeros(3,1);  % initialize w to be a matrix of zeros

for i=1:size(A,1) % loop to assign element i of vector w
    for j=1:length(v)  % loop to perform the sum over j
       w(i)=w(i)+A(i,j)*v(j);  % add A(i,j)*v(j) to the current value of w(i)
    end
end
w   %display w
```

Note in the code above that you could have simply said “for i=1:3” and “for j=1:2”. The above code is written so that it could more easily be generalized to multiplying matrices and vectors of different sizes.

4. *Finding prime numbers*

```
clear all;  %clear all variables
close all;  %close all figures

%parts (a),(b), & (c)
for N = 2:30  %can set to N=22:22 if want to check just the number 22

    PrimeFlag = 1; %this will be changed to zero if N is found
    %not to be a prime number
    for divisor = 2:round(N/2)
        if (mod(N,divisor)==0) %note use of double equal sign
            PrimeFlag = 0; %Is not prime
            break; %for efficiency: once a divisor is found, we know this
                   %value of N isn't prime, so we don't need to test
                   %any more divisors
        end
    end %for divisor loop

    if PrimeFlag
        disp([int2str(N) ' is prime'])
    else  %is not prime, i.e. PrimeFlag must be zero
        disp([int2str(N) ' is not prime'])
    end

end %for N loop
```

5. *Numerical derivatives*

i)

```
>> dt = 0.1
>> t = -1:dt:1
```

ii) `x = sin(2*pi*t)`

The 7th element of x is x(7) = sin(2*pi*t(7)) = sin(2*pi*(-0.4)) = -0.5878.

iii)

```
% program to take the derivative of the function sin(t)
% note how the variable names help to make the code clear
clear all;
dt = 0.01;   % small time step to represent Delta_t in derivative formula
t = -1:dt:1;  % time vector
x = sin(2*pi*t);  % the function to be differentiated

dx = diff(x);  % a vector containing the differences between the x points

dxdt = dx/dt;  % the derivative

%plotting
figure(1)   % the function
plot(t,x)
figure(2)   % the derivative
plot(t(1:end-1),dxdt)  % note: last element removed from t vector
```

6. *Computing the value of an integral*

```
% Exercise 4: Computing an integral

dx = 0.001;  %width of rectangles in computing derivative
x = 1:dx:3;  %vector containing values of x at which to compute derivative
xsquared = x.^2; %vector containing square of each element of vector x

Integral = dx*sum(xsquared) %integral formula
```